Fall 2020

Exploring Optimization Opportunities in Non-Volatile Memory Systems

Prakhar Bansal
Iowa State University

Exploring Optimization Opportunities in Non-Volatile Memory Systems

by

Prakhar Bansal

A Creative Component submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Dr. Mai Zheng, Major Professor

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this creative component report. The Graduate College will ensure this report is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2020

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this creative component report. First and foremost, Dr. Mai Zheng for his guidance, patience and support throughout this research and the writing of this report. His insights and words of encouragement have often inspired me and renewed my hopes to be successful in my research work and also completing my graduate education. I would also like to thank my colleague Om Gatla for his support, efforts and contributions to this work.

ABSTRACT

Modern storage systems utilize Non-Volatile Memories (NVMs) to reduce the performance and density gap between memory and storage. NVMs are a broad class of storage technologies, including flash-based SSDs, Phase Change Memory (PCM), Spin-Transfer-Torque Random Access-Memory (STTRAM). These devices offer low latency, fast I/Os, persistent writes, and large storage capacity compared to volatile DRAM. However, researchers are still working on the possibility of building systems that can leverage these NVMs to deliver low latency and high throughput to applications. Conventional systems were designed to persist data on hard drives, which has higher latency than NVM devices. Hence, in this work, we intend to explore opportunities to improve performance and reliability in the NVM based systems.

One class of NVM devices that are placed on the memory bus is Persistent Memory (PM). Examples of PM technologies include 3D XPoint, NVDIMMs. Applications need to be modified to use the PM devices, which requires a lot of human effort and could lead to programming errors. Hence, reliability is also necessary to build systems to utilize the PM. Additionally, as persisted data is expected to be recoverable systems in case of a crash, PM applications are responsible for providing that reliability support at the application level instead of relying on the file system.

In this work, we evaluate the performance of popular key-value store RocksDB that is optimized for flash storage and also the reliability guarantees provided by recent works, which provides the testing framework for determining crash-consistency bugs in PM systems. Based on this analysis, we also present some opportunities to optimize performance and reliability in NVM systems.

CHAPTER 1. OVERVIEW

In this chapter, we will briefly introduce our project, the Motivation behind it, and our research goals. Finally, we will summarise the structure of this report.

1.1 Introduction

Modern applications such as Facebook, Linked In, etc., use key-value stores like RocksDB [24][13]and LevelDB [7] as a storage engine for data management or as a back-end for stateful services [15]. Examples of services where RocksDB is used as a back-end by Facebook include MyRocks [33], MongoRocks [9]. These key-value stores are highly dependent on DRAM to deliver desired performance, even when using flash for data storage. Since DRAM is considered 1000x faster than flash [25], the above systems can cache hot indices and objects on DRAM. Facebook is extensively using MySQL database in their data center environments, and they implemented MyRocks that uses RocksDB as back-end storage engine. MyRocks utilizes a large amount of DRAM to deliver high throughput and low latency to the applications. Therefore, exploiting non-volatile memories (NVMs) in hybrid systems is widely considered a promising mechanism to deliver performance to applications.

A high-performance NVM that is places on memory bus and accessible with byte-addressability via a load/store interface is known as Persistent Memory (PM) [31][18]. PM allows the software to bypass OS indirections and not impacted by the possible latencies associated with page caching. Hence, it can directly update persistent data in memory and lead to better performance than traditional systems. However, it brings the challenge of providing reliability support which was handled by file system in conventional systems.The data on PM should be recoverable in case of crash. Now, PM systems need to provide the crash-consistency support at the application level only. Ensuring crash-consistency generally have two requirements:

Durability and Ordering. Durability requires that write should become persistent in PM and ordering means that one write becomes persistent before another. To provide these crash-consistency guarantees in PM systems, Intel introduced two new instructions in x86 Instruction Set Architecture (ISA); **clwb** and **sfence**. Also, transactional libraries(e.g., Intel's PMDK [11]) using these low-level instructions were built for applications to manage data on persistent memory.

However, even with the help of transactional libraries, developers need to understand the specifications of crash-consistency guarantees provided by these libraries. Hence, it is hard for developers to identify, whether crash-consistency algorithm is correctly implemented or not. To overcome this problem, testing frameworks [31][30] are developed to evaluate the crash-consistency guarantees in software developed for PM including Intel's PMDK. These frameworks still require lot of manual effort in annotating the source code and generating the test suites, so we would like to explore potential improvements or optimal methodology to evaluate the crash-consistency guarantees to improve the reliability support in PM systems.

1.2 Motivation

Popular flash-based key-value stores consume a large amount of DRAM to provide high-performance database operations. For example, MyRocks [10], which is built on top of RocksDB, is a MySQL database in Facebook and primarily used for data center environments [25]. It [10] uses 96GB of DRAM with flash as persistent storage to provide high throughput and low latency to applications. Other key-value stores such as Memcached [34], Tao[20] are also highly dependent on DRAM. However, accommodating large DRAM for those databases can be challenging because DRAM can be expensive for data center providers, primarily due to an increase in DRAM costs due to limited supply globally. Additionally, increasing DRAM only would not necessarily improve the performance of NVM systems. The factors such as cell sizes, power consumption, and DIMM slot availability prevent system performance from being further improved via increasing DRAM size [19][25]. Therefore, in this work we explore the opportunities

to optimize the performance of popularly used embedded key-value store RocksDB using the current and emerging NVMs.

In addition to performance, NVM systems based specifically on persistent memory (PM) also need to provide reliability support at the application level that was traditionally the file system's responsibility. In general, reliability is defined by crash-consistency guarantees in PM systems. Recent works such as PMTest [31], XFDetector [30], developed state of the art testing frameworks to evaluate the crash-consistency guarantees in the software developed for the popular PM systems. These works detected bugs in software developed by expert programmers, including an optimized file system (PMFS) and PM systems based on transactional library PMDK[11]. They were also able to detect bugs in real-world applications such as Redis [28], Memcached. In this work, we explore the tool XFDetector that can identify crash-consistency due to cross-failure interactions.

1.3 Research Goals

In our research work, we aimed at accomplishing following goals:

G1: **Exploring Opportunities for Performance Optimization of NVM systems**:

By performing an analysis of existing NVM systems, we intend to explore the opportunities to optimize or develop systems that can deliver high performance to applications.

G2: **Exploring Opportunities for Reliability Optimization in Persistent Memory applications**:

We also intend to explore opportunities for reliability optimization in PM systems after evaluating the support by the existing works that provide a testing framework to evaluate the crash-consistency guarantees in PM applications.

CHAPTER 2. Performance Optimization Opportunities in Non-Volatile Memory Systems

This chapter contains five sections. The First section focuses on background knowledge and earlier works on performance optimization in RocksDB with non-volatile memories. The second section presents our analysis methods for performance analysis in RocksDB. Later, we will also discuss our experimental results and performance optimization opportunities in RocksDB with emerging Non-volatile memories.

2.1 Literature Review

This section will discuss the background on the key-value store RocksDB [24][13], which is based on the Log-Structured Merge (LSM) tree [35] data structure. It also discusses the related work done for the performance optimization in RocksDB with non-volatile memory (NVM) technologies.

2.1.1 RocksDB

RocksDB is an embedded key-value store developed at Facebook and used as a back-end storage engine by external applications such as Linked In, Microsoft, etc. It is built on top of LevelDB [7] to scale it on servers with multiple cores to use fast storage, and it is written mostly in C++. RocksDB has more features than LevelDB and thus provides a significant performance improvement over it. Some of the features include bloom filters, block cache, tunability, etc. It has both memory and storage components that are described below:

1. **Memory components**

   These components provide fast access to user queries because they keep their data in memory.

   (a) **MemTable**: It is a skiplist data structure that can temporarily host incoming writes. New writes inserts data to MemTable, and reads are first queried from MemTable before reading from data files on persistent storage because data in MemTable is newer. After a specific predetermined threshold size, this MemTable becomes immutable and replaced by a new one.

   (b) **Block Cache**: RocksDB caches uncompressed data blocks in Block cache for fast lookups. In RocksDB's optimization of the LSM tree, recently accessed data blocks of SST files are stored in block cache, so access to recently fetched data need not result in I/O operations. Users can pass in a Cache object to a RocksDB instance of the block cache's desired capacity size. It allows users to control the block cache size as the cache object can be shared by multiple RocksDB instances [3].

2. **Storage Components**:

   (a) **Write Ahead Log(WAL)**: This is the transaction log in RocksDB for recovery purposes. Every incoming write in RocksDB is first written to WAL and then flushed synchronously to persistent storage such as a disk. WAL is used to recover the data in the MemTable in case of failure. This way, the database RocksDB can restore the database to the original state.

   (b) **Sorted Sequence Tables (SST) files**: These are the data files on persistent storage stored in levels of increasing size.

- **Why RocksDB?**: We looked into a few other key-value stores and in-memory databases initially in our research, but we planned to explore RocksDB in detail for a few reasons. Firstly, applications in general use remote procedure calls to access their data over a network, introducing high latency. Since RocksDB is optimized for flash storage,

applications can maintain their data directly on flash instead of accessing data over the network. Secondly, RocksDB is quite flexible as it allows the tuning of different parameters to manage its resource consumption and optimize it for performance.

- **RocksDB Architecture**: RocksDB is based on the Log-structured merge tree (LSM), which has been widely adopted in storage layers of modern NoSQL systems, including Cassandra [1], LevelDB [7], RocksDB. Unlike traditional index structures like B+ tree that apply in-place updates, LSM tree buffer writes in memory, merge them and flush them to disk. Whenever a request for write is sent to the LSM tree, it is added to MemTable, an in-memory write buffer implemented as a skip-list data structure with time complexity of O(logn) inserts and searches. Before writing to MemTable, the data is first appended to a write-ahead log (WAL) for recovery. When the MemTable reaches a predetermined size, then the current WAL and MemTable become immutable, and a new WAL and MemTable are allocated for subsequent writes. The MemTable contents are flushed to the "Sorted Sequence Table (SST) data file on a persistent storage medium, and upon completion, the WAL and MemTable containing the data just flushed are discarded. This design brings several advantages like superior write performance, effective space utilization, and new writes that can be processed concurrently to an older MemTable's flushing. On top of that, write throughput for flash storage is order-of-magnitude higher than alternatives, and optimizations have been done to RocksDB to exploit such storage mediums even better. These advantages have enabled LSM trees to serve a large variety of workloads. As reported by Facebook, RocksDB is used a storage engine for the applications such as real-time processing [22], graph processing [2], and Online Transaction Processing (OLTP) workloads [10].

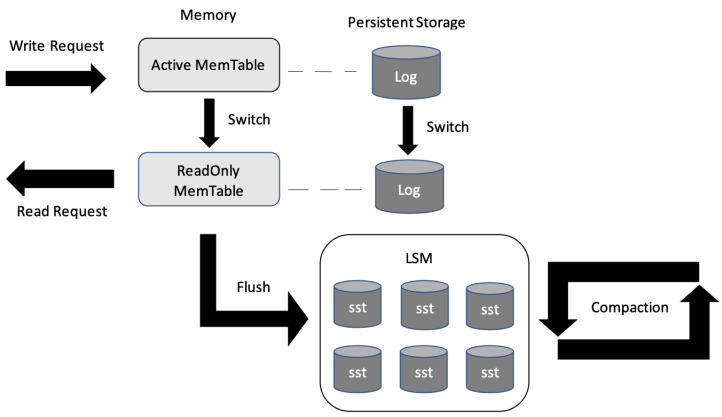

Figure 2.1 RocksDB Architecture [14]

- **Compaction Mechanism in LSM Tree**: SST files in the LSM tree are immutable, so we cannot directly update a key-value pair and pose a potential storage problem. If one has to store all the old data, the LSM tree handles this issue by doing compactions.

  As mentioned above, the contents of MemTable are flushed to the SST data file. In each of the SST files, sorted data is stored in unaligned 16KB blocks (when uncompressed). Each SST also has an index block for binary search with one key per SST block. These SST files are organized into a sequence of exponentially increasing size levels, where each level can have multiple SST files, as shown in Fig.2.1. Level-0 is treated differently because SST files in that level probably have overlapping key ranges while SST files in higher-level have non-overlapping key ranges. After the number of files in Level-0 exceeds a predetermined threshold the level-0 SST files are merged with level-1 SST files with overlapping key ranges. This merging process is known as compaction because it generates new SST files with overwritten keys, and deleted keys will be removed from output files. If the write rate to the database is high, more resources would be required for computation because the LSM tree would be in good shape and remain stable only if compaction can keep up with new writes to the database.

Figure 2.2 Format of Sorted Sequence Table [25]

Leveled Compaction: The default Compaction strategy in RocksDB is leveled compaction. Leveled compaction organizes the files on multiple levels, usually with increasing sizes. This process ensures that at any given time, each SST files in a level will contain at most one entry for any given key and snapshot. The I/O that occurs during compaction is efficient as it only involved bulk reads and writes of entire files [24].

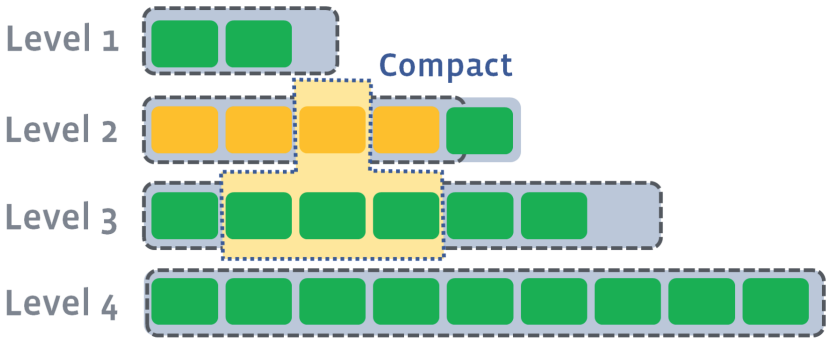

Figure 2.3 Leveled Compaction [4]

For the read request, key lookup occurs at the successive level until the key is found, or it is found that the key is not present in even the last level [33]. The search begins with MemTables, followed by level-0 SST files and then the SST files in successive levels. As these SST files are sorted, the whole binary search is used at each level, and this maximizes the read performance in RocksDB but at the cost of write amplification. The First search locates the target SST using the metadata in the manifest file, then the second search locates the target data block with the SST using the index block, and the final search looks for the key within the data block. Bloom filters are kept in each SST file and used to eliminate unnecessary search within an SST file.

2.1.2 Non-volatile Memory

Non-volatile Memory (NVM) is persistent storage technology with the potential of replacing DRAM in both a data center and consumer use cases[13]. NVM is available in two form factors: DIMM form factor, which is byte-addressable, and as a PCIe or SATA block interface access through block device. A byte-addressable NVM is also known as Non-volatile main memory (NVMM) or Persistent memory (PM). Examples of such persistent memories are 3D XPoint [5][6], NVDIMMs, etc. that offer high density than DRAM with comparable latency and bandwidth. Bypassing the storage stack and directly accessing NVMM is essential for leveraging the performance benefits.

2.1.3 Related work

Several projects optimized RocksDB or state of the art LSM tree applicable to RocksDB using emerging NVMs. NoveLSM [27] proposes an immutable MemTable on PM between DRAM and disk component. Also, there is a mutable MemTable to directly update data in PM and used along with DRAM MemTable to reduce stalls due to compaction. In another recent work, MatrixKV [36] proposes a matrix container to manage level L0 of the LSM tree. It increases each level's width in the LSM and reducing the number of levels in LSM tree. These design choices reduce write stalls and write amplification in RocksDB's LSM tree.

2.2 Methodology

This section will discuss our methodologies for analyzing the performance of popular KV store RocksDB, as it intends to utilize the fast DRAM and persistent SSDs to provide high-performance database accesses. It also includes the methods for analyzing the reliability of persistent memory applications using the cross failure bug detection mechanism.

2.2.1 Tracking Memory Usage of RocksDB

- **Purpose**: As mentioned earlier, applications that serve requests from memory have high memory requirements to provide desired performance. As Non-volatile memory (NVM) such as Intel Optane DC persistent memory [6] provides slower denser DRAM, we want to generate use cases with RocksDB that requires large DRAM to deliver high performance to applications. Hence, here we tried to exploit the memory usage in RocksDB while benchmarking it with YCSB [23][17].

- **Memory Components in RocksDB**: There are a few components in RocksDB that contribute to memory consumption. These are Block Cache, MemTables, Indexes, and bloom filters, and blocks pinned by iterators. In our methodology, we focus on tuning MemTables and block cache only as in initial observation and study; we find that these are significant contributors to memory usage in RocksDB. Three parameters can be configured to control memory usage in RocksDB. Except for block_cache_size, the other two parameters can be directly tuned from the RocksDB's Options file A.3:

  1. Size for a MemTable in RocksDB can be changed by **the write_buffer_size.**
  2. Size of Block cache can be changed by **block_cache_size**.
  3. The parameter that decides the maximum number of MemTables held in memory before RocksDB would flush them to the local disk as SST files is **max_write_buffer_number**.

- **Tuning MemTable size** As we explained in the previous chapter that in RocksDB, write is written into the memory buffer called MemTable. In this method, we started with the default configuration of RocksDB, where the size of the MemTable is 128MB. As RocksDB supports tuning of several of its parameters, we tune the MemTable size in our experiments of benchmarking YCSB with RocksDB.

  Here, we kept on increasing the size of MemTable because we want to track the memory requirements in RocksDB and analyze the impact on RocksDB's performance with increasing memory usage.

- **Tuning Block Cache size**: In a default configuration, RocksDB will use LRU-based block cache implementation with capacity of 8MB. To set a customized block cache, we modify the source code A.1 of RocksDB's client for YCSB, where we called the NewLRUCache() method to create a cache object and set it to block-based table options. After modifying the code in RocksDBClient.java, YCSB has to be compiled again. The following code snippet shows the changes performed for configuring the block cache. After configuring the block cache, we analyzed the memory usage and the impact on performance with different block cache sizes by passing different values to the new LRUCache().

2.2.2 Benchmarking RocksDB with YCSB

- **Purpose**: To test and analyze the performance of RocksDB, we ran it on standard benchmarking tool YCSB, since YCSB's generates workloads that are considered close to real-world ones. YCSB can generate queries with similar statistics for a given query type ratio, KV-pair hotness distribution, and value size distribution as those in realistic workloads [37].

- **Method**: We used YCSB as a benchmarking tool to evaluate RocksDB's performance when data resides on flash storage. Researchers usually consider the workloads generated by YCSB to be close to real-world workloads. YCSB can generate queries with similar statistics for a given query type ratio, KV-pair hotness distribution, and value size

distribution as those in realistic workloads [37]. In this method, we focused on YCSB's workload A(50% Reads, 50% writes) with Zipfian distribution since it is an update intensive workload. We have used RocksDB 6.2.2 in our analysis, the latest supported version in YCSB's Github project..

Since RocksDB is an embedded key-value store and we are benchmarking it with YCSB, we can only track the memory usage for the YCSB process. We utilize the Linux utility "**ps**" which stands as an abbreviation for "Process Status" to track the physical and virtual memory utilization for a running YCSB process. We wrote the shell script A.2 to monitor the memory usage by the YCSB process.

2.2.3 Virtual Machine Server

- **Purpose**: In cloud systems, one physical machine needs to host multiple Virtual Machines (VMs) and large memory capacity may allows hosting more VMs with better efficiency. Therefore, we tried to determine the correlation between memory capacity and QEMU-KVM Virtual Machines performance through our experiments.

- **Performance of RocksDB with VMs**: Here, we evaluated the performance of RocksDB when running its instances inside multiple virtual machines parallelly on a single VM server. Also, we investigated the memory requirements of virtual machines in order to explore opportunities to improve the performance of the application running inside it. We computed the **"Normalized Average run time"** for the RocksDB's instances inside the multiple virtual machines. For our work, we used the virtual machines with following sizes:

  1. DRAM of 5GB
  2. DRAM of 10GB

- **Host Swap Space usage**: Here, we would like to track the swap space usage by the QEMU-KVM virtual machines, if there is no or limited memory available on the system for running user processes. Initially, when the total swap space available on system was only

8GB, we observed significantly less swap space by virtual machines even after the system's memory is exhausted. Hence, we increase our system's swap space to 108GB to check if more virtual machines can be created and accessed. We used the Linux utility **"VmSwap"** to find out the swap memory usage used by the QEMU-KVM process for each VM.

2.3 Experiments and Results

This section will discuss the experimental setup and results based on our methodology in the previous chapter.

- **System Specifications**

  1. **CPU**:Intel Xeon E-2174G, 3.8GHz, 8 cores
  2. **DRAM**: 64GB
  3. **OS**: Ubuntu 18.04, linux kernel 5.1.0

- **SSD Specifications**

  1. **SSD**: KXG50ZNV512G NVMe TOSHIBA 512GB
  2. **Sequential read**: upto 3000MB/sec, **Sequential write**: up to 2100MB/sec

2.3.1 Tracking Memory Usage in RocksDB

This section will demonstrate our results with memory tracking in RocksDB, while we tuned RocksDB's paramters responsible for memory usage. We also studied the variation in performance of RocksDB with rise in memory usage.

- **Workload Configuration to run YCSB on RocksDB**:

  1. YCSB Record size = 280 bytes(key = 24, value = 256)
     No. of records = 100 Million, Workload = 50%Reads and 50% writes
  2. Initial MemTable size = 128MB(default)

3. Initial Block Cache size = 8MB(default)

- **Tuning Memtable in RocksDB** We change the size of MemTable by tuning the parameter; write_buffer_size from RocksDB's options file. Here, the block cache remains unchanged when we tuned the MemTable.

MemTable(GB)	Physical Memory(GB)	Virtual Memory(GB)
0.12	0.72	21.4
0.5	2.29	22.72
1	4.02	24.99
8	15.90	41.29
12	15.83	48.75
16	15.82	40.21

Table 2.1 Memory usage with Increasing MemTable size

Table 2.1 show the physical and virtual memory usage in RocksDB for the different MemTable sizes.

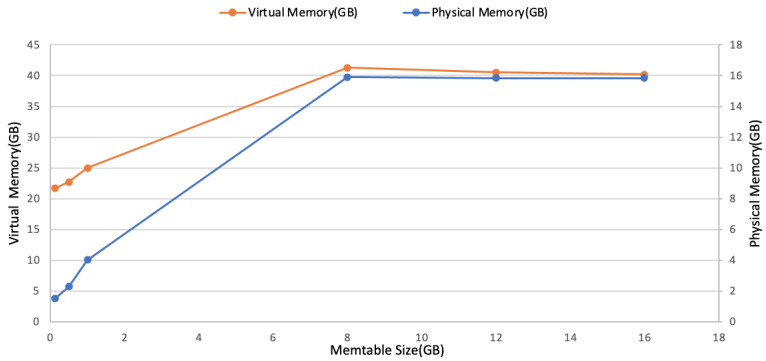

Figure 2.4 Memory tracking in RocksDB

Figure 2.4 shows the variation in virtual and physical memory usage by the YCSB's java process when tuned the MemTable size in the RocksDB options file. We observed that memory usage becomes constant after increasing up to a specific MemTable size.

MemTable(GB)	RunTime(secs)
0.12	2616
0.5	2283
1	2030
8	1925
12	1880
16	2472

Table 2.2 YCSB's stress testing(run phase) with various MemTable sizes

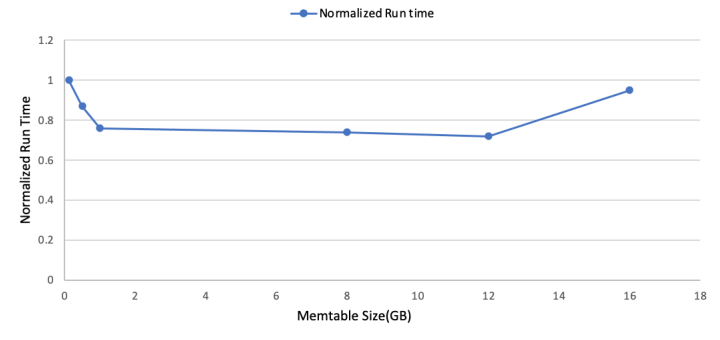

Figure 2.5 YCSB's Stress testing for RocksDB

Figure 2.5 demonstrates that runtime during stress test decreases with the MemTable size of 12GB. However, runtime increases after MemTable of 16GB, and we tried to determine the reason for this behavior in section 2.4.

- **Tuning Block Cache in RocksDB**: For experiments with tuning the block cache, MemTable size was kept at 16GB, and we begin with a default block cache size of 8MB and increase it up to 16GB, as we did not observe a significant change in memory usage beyond 16GB. Also, per our knowledge, 16GB is the generally maximum size of the block cache size used for the Facebook's production use cases such as MyRocks.

Block Cache(GB)	Physical Memory(GB)	Virtual Memory(GB)
0.008	15.47	35.70
0.5	15.48	35.70
1	15.47	35.67
4	15.46	35.70
8	16.47	38.68
16	24.76	46.87

Table 2.3 Memory usage with Increasing Block Cache size

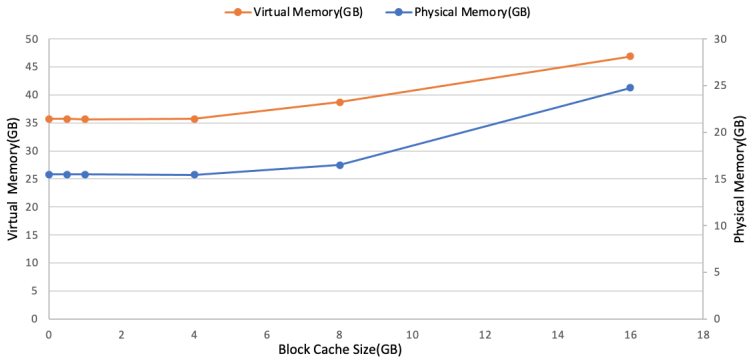

Figure 2.6 Memory Usage in RocksDB

On increasing the block cache size in RocksDB, we observed the gradual increase in memory usage by the YCSB's java process after the block cache size of 4GB.

Block Cache(GB)	RunTime (secs)
0.008	1904
0.5	1880
1	1855
4	1847
8	1785
16	1697

Table 2.4 Stress testing Runtime with various Block Cache sizes

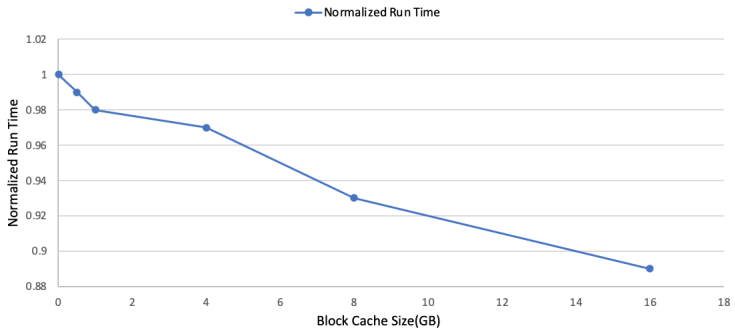

Figure 2.7 YCSB's stress test phase for RocksDB

As seen in figure 2.6, When we tune the block cache in RocksDB, there was a rise in physical and virtual memory usage by YCSB's java process. The increase in memory usage also improves the performance of RocksDB, which can be seen in figure 2.7. We did not observe an improvement in RocksDB's performance beyond the block cache size of 16GB.

2.3.2 RocksDB on each VM

The following system and YCSB's workload configuration were used to generate results in this section:

- **One Physical VM Server**
  1. Intel Xeon E-2174G, 64GB DRAM
  2. 8 - 108GB Swap Space on Hard disk drive
  3. Disk Space for each VM = 20GB
- **Two types of VM**
  1. VM with 5GB DRAM
  2. VM with 10GB DRAM

- **Running YCSB on RocksDB in each VM**

  1. RocksDB's MemTable size = 2GB
  2. YCSB record size = 280 bytes(key = 24, value = 256)

     No. of records = 20 Million, No. of threads = 4 Workload = 50%Reads and 50% writes

No. of VMs	Average RunTime:5GB DRAM	Average RunTime:10GB DRAM
1	202	165
2	222	203
5	412	316
10	2161	up to 10x

Table 2.5 Average RunTime(secs) for RocksDB running in virtual machines

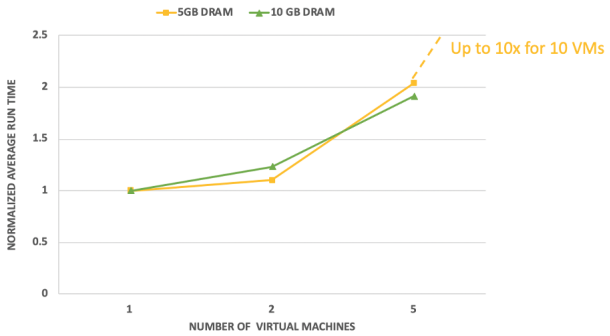

Figure 2.8 Normalized Average run time of YCSB-RocksDB instances within VMs

From figure 2.8, We can see that the average run time of workload increases with the VM count. For VMs with 5GB DRAM, up to 12 VMs were successfully supported but could not run applications on the 11th and 12th VM. For VMs with 10GB DRAM, up to 8 VMs were supported; however, we could not run any application inside the 8th VM.

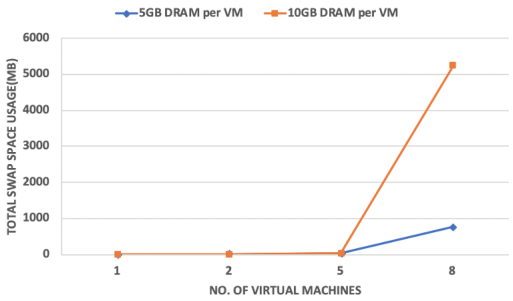

Figure 2.9 Total Host's swap space usage by VMs

From figure 2.9, the host's swap space usage was increased after the number of virtual machines exceeded the count of 5. Also, even with enough swap space of 108GB, we observe that VMs may still fail to launch when the system's memory has been exhausted for the user processes. Hence, these results show that memory capacity is critical for VM server and VMs performance.

2.4 Performance Debugging in RocksDB

From the results generated from the above experiments, we tried to find performance bottlenecks in RocksDB using utilities provided in RocksDB.

2.4.1 Analyzing RocksDB statistics

RocksDB generates some statistics on persistent storage for analysis of its database operations.

In figure 2.5 , we see that run time was increased after MemTable size further from 12GB to 16GB. This analysis compared the statistics reported in the LOG file by RocksDB for MemTable 8GB and 16GB. Since the difference between run time with MemTable sizes of 8GB and 12GB is not significant, it suffices to understand the pattern in RocksDB's performance.

```
** Compaction Stats [usertable] **
Level    Files   Size     Score Read(GB)  Rn(GB) Rnp1(GB) Write(GB) Wnew(GB) Moved(GB) W-Amp Rd(MB/s) Wr(MB/s) Comp(sec) CompMergeCPU(sec) Comp(cnt) Avg(sec
) KeyIn KeyDrop
----------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------
  L0      0/0    0.00 KB   0.0      0.0     0.0      0.0      27.8     27.8       0.0   1.0      0.0    345.1     82.48              0.00         3   27.494
       0       0
  L1      8/0   510.16 MB   1.0     28.3    27.8      0.5      27.8     27.3       0.0   1.0    120.0    117.7    241.43            143.18         1   241.43
4    101M  1768K
  L2     91/0    4.98 GB   1.0     34.6    27.3      7.4      30.0     22.7       0.0   1.1     17.9     15.6   1976.04            512.18       199    9.930
    124M    16M
  L3    443/0   27.60 GB   0.6     61.8    21.9     39.9      39.9      0.0       0.7   1.8     27.5     17.8   2299.38            682.64       347    6.626
    223M    79M
 Sum    542/0   33.09 GB   0.0    124.7    77.0     47.8     125.5     77.8       0.7   4.5     27.8     27.9   4599.34           1338.00       550    8.362
    450M    97M
 Int      0/0    0.00 KB   0.0      0.0     0.0      0.0       0.0      0.0       0.0   0.0      0.0      0.0      0.00              0.00         0    0.000
       0       0

** Compaction Stats [usertable] **
Priority    Files   Size     Score Read(GB)  Rn(GB) Rnp1(GB) Write(GB) Wnew(GB) Moved(GB) W-Amp Rd(MB/s) Wr(MB/s) Comp(sec) CompMergeCPU(sec) Comp(cnt) Avg(
sec) KeyIn KeyDrop
----------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------
 Low      0/0    0.00 KB   0.0    124.7    77.0     47.8      97.7     50.0       0.0   0.0     28.3     22.2   4516.86           1338.00       547    8.258
    450M    97M
User      0/0    0.00 KB   0.0      0.0     0.0      0.0      27.8     27.8       0.0   0.0      0.0    345.1     82.48              0.00         3   27.494
       0       0
Uptime(secs): 3460.4 total, 0.0 interval
Flush(GB): cumulative 27.794, interval 0.000
AddFile(GB): cumulative 0.000, interval 0.000
AddFile(Total Files): cumulative 0, interval 0
AddFile(L0 Files): cumulative 0, interval 0
AddFile(Keys): cumulative 0, interval 0
Cumulative compaction: 125.51 GB write, 37.14 MB/s write, 124.73 GB read, 36.91 MB/s read, 4599.3 seconds
Interval compaction: 0.00 GB write, 0.00 MB/s write, 0.00 GB read, 0.00 MB/s read, 0.0 seconds
Stalls(count): 0 level0_slowdown, 0 level0_slowdown_with_compaction, 0 level0_numfiles, 0 level0_numfiles_with_compaction, 0 stop for pending_compaction_byt
es, 202 slowdown for pending_compaction_bytes, 0 memtable_compaction, 0 memtable_slowdown, interval 0 total count
```

Figure 2.10 RocksDB's Statistics when running with MemTable of 8GB

If we compare the figure 2.10 and 2.11, we can see from $L0- > L1$ compactions- comp(sec) are slow in RocksDB with MemTable size of 16GB. Also, read, write bandwidth with 16GB MemTable is less than compared to 8GB MemTable. We are focusing on $L0- > L1$ because when compacting $L0- > L1$, compaction includes all files from L1 and with all files from L1 getting compacted with L0, compaction $L1- > L2$ cannot proceed; it need to wait for the L0-¿L1 compaction to finish.

```
** Compaction Stats [usertable] **
Level    Files   Size     Score Read(GB)  Rn(GB) Rnp1(GB) Write(GB) Wnew(GB) Moved(GB) W-Amp Rd(MB/s) Wr(MB/s) Comp(sec) CompMergeCPU(sec) Comp(cnt) Avg(sec
) KeyIn KeyDrop
----------------------------------------------------------------------------------------------------------------------------------------------------------
----------------
  L0      0/0    0.00 KB   0.0      0.0     0.0      0.0      27.8     27.8       0.0   1.0      0.0    331.3     85.86              0.00         2   42.932
       0       0
  L1      7/0   451.91 MB   0.9     27.8    27.8      0.0      27.8     27.8       0.0   1.0    113.2    113.1    251.21            138.38         1   251.20
9    100M       0
  L2     80/0    4.97 GB   1.0     27.3    27.3      0.0      27.2     27.2       0.0   1.0     16.5     16.4   1698.07            442.81       434    3.913
     98M       0
  L3    354/0   22.22 GB   0.4      0.0     0.0      0.0       0.0      0.0      22.2   0.0      0.0      0.0      0.00              0.00         0    0.000
       0       0
 Sum    441/0   27.63 GB   0.0     55.1    55.1      0.0      82.7     82.7      22.2   3.0     27.7     41.6   2035.14            581.18       437    4.657
    198M       0
 Int      0/0    0.00 KB   0.0      0.0     0.0      0.0       0.0      0.0       0.0   0.0      0.0      0.0      0.00              0.00         0    0.000
       0       0

** Compaction Stats [usertable] **
Priority    Files   Size     Score Read(GB)  Rn(GB) Rnp1(GB) Write(GB) Wnew(GB) Moved(GB) W-Amp Rd(MB/s) Wr(MB/s) Comp(sec) CompMergeCPU(sec) Comp(cnt) Avg(
sec) KeyIn KeyDrop
----------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------
 Low      0/0    0.00 KB   0.0     55.1    55.1      0.0      54.9     54.9       0.0   0.0     28.9     28.9   1949.28            581.18       435    4.481
    198M       0
User      0/0    0.00 KB   0.0      0.0     0.0      0.0      27.8     27.8       0.0   0.0      0.0    331.3     85.86              0.00         2   42.932
       0       0
Uptime(secs): 3489.3 total, 0.0 interval
Flush(GB): cumulative 27.780, interval 0.000
AddFile(GB): cumulative 0.000, interval 0.000
AddFile(Total Files): cumulative 0, interval 0
AddFile(L0 Files): cumulative 0, interval 0
AddFile(Keys): cumulative 0, interval 0
Cumulative compaction: 82.72 GB write, 24.27 MB/s write, 55.09 GB read, 16.17 MB/s read, 2035.1 seconds
Interval compaction: 0.00 GB write, 0.00 MB/s write, 0.00 GB read, 0.00 MB/s read, 0.0 seconds
Stalls(count): 0 level0_slowdown, 0 level0_slowdown_with_compaction, 0 level0_numfiles, 0 level0_numfiles_with_compaction, 0 stop for pending_compaction_byt
es, 0 slowdown for pending_compaction_bytes, 0 memtable_compaction, 0 memtable_slowdown, interval 0 total count
```

Figure 2.11 RocksDB's Statistics when running with MemTable of 16GB

2.4.2 RocksDB Tuning Advisor

RocksDB's project also provides the command-line tool to advise for tuning the RocksDB for performance. Users need to provide the RocksDB's LOG and options file as the inputs to the tool. We use this tool to know the bottlenecks with our RocksDB's configuration leading to a higher run time when benchmarking RocksDB with YCSB. We used the tuning advisor following the below steps, after running Rocksdb with a MemTable size of 16GB.

cd rocksdb/tools/advisor

python3 -m advisor.rule_parser_example –rules_spec=advisor/rules.ini
–rocksdb_options=/tmp/ycsb_16gb/OPTIONS-000015
–log_files_path_prefix=/tmp/ycsb_16gb/LOG –stats_dump_period_sec=20

```
Rule: level0-level1-ratio
OptionCondition: level0-level1-ratio options: ['CFOptions.level0_file_num_compaction_trigger', 'CFOptions.write_buffer_size', 'CFOptions.max_bytes_for_level_base'] expression: int(options[0])*int(optio
ns[1])-int(options[2])>=1 trigger: {'default': ['2', '17179869184', '536870912'], 'usertable': ['2', '17179869184', '536870912']}
Suggestion: inc-base-max-bytes option : CFOptions.max_bytes_for_level_base action : increase
scope: col_fam:
{'usertable', 'default'}
```

Figure 2.12 Output from Tuning Advisor

Figure 2.12 show that tuning advisor suggested to increase the size of Level 1(max_bytes_for_level_base) in RocksDB. Because smaller size of L1 slows down the write bandwidth to L1, it eventually leads to slower compaction rate in RocksDB between $L0 -> L1$ and $L1 -> L2$.

2.5 Opportunities to Optimize Performance of RocksDB with current and emerging NVMs

This section discussed the potential improvements and performance optimization opportunities in RocksDB with Non-Volatile Memory devices.

- **Tuning RocksDB**

  RocksDB provides several options that can be tuned for achieving better performance. Here, we provide only a few options that can be configured for performance improvement in RocksDB.

  1. **Format version in block table**

     One of the parameters in block-based table options in RocksDB's options file is format_version. The default value of format_version is 2. If the value is changed to 4, it will significantly reduce the index block size, which frees more space in the block cache [13]. This change would result in a better hit rate in block cache for data and filter blocks.

  2. **Increase maximum number of subcompactions**

During the compaction process between two levels, by default, only one compaction thread runs that merge files from level N with level N +1 (N = 0,1,2...). If we increase the value of the maximum number of subcompactions for each level, the compaction rate would be higher overall, which will improve the performance in RocksDB.

- **Large Address space using Persistent memory**

  One type of non-volatile memory that is available through memory bus is Persistent Memory (PM) such as Intel Optane "data center" persistent memory. PM will be placed on a memory bus like DRAM and can be accessed via processor loads and stores without the interference of software [26]. As we can interpret from the above results, RocksDB utilizes a large address space to deliver high performance. Although PM has higher latency than DRAM, it can provide larger virtual address space to the applications.

- **Reducing Write Amplification in RocksDB**

  The amount of data written to flash storage compared to the amount of data the application wrote is known as Write Amplification. Although persistent memory can provide lower latency and higher read/write bandwidth than SSD, but write amplification due to the compaction among LSM tree levels would still be impacting the performance of RocksDB and PM's write endurance. Inspired from the works [32][12], moving the LSM tree on PM can reduce the write amplification by separation of key and values on PM, where values would be stored in separate log which can be a ring buffer or FIFO queue. With this method, values doesn't need to be written down to lower levels of LSM during the compaction process that is responsible for write amplification in RocksDB's LSM tree.

CHAPTER 3. Reliability Optimization Opportunities in Persistent Memory Systems

This chapter discusses the background and related work on evaluating the reliability support provided by software developed for persistent memory systems. It also discusses our experimental methodology to use the state of the art testing framework XFDetector [30] and reproduce the bugs detected by it in the common PM systems. Finally, we will also present a few opportunities for optimizing the reliability support for PM systems.

3.1 Literature Review

This section covered the background on reliability in Persistent Memory (PM) aware applications in which we will introduce programming for PM systems and its challenges. We will also discuss related work that developed the testing framework to identify the persistency bugs in crash-consistent software for PM applications.

3.1.1 Persistent Memory Applications

The emergence of PM technologies such as 3D XPoint has led to an increase in applications that can utilize PM because applications can manipulate data on PM directly. Since data on the PM have to be crash recoverable, crash-consistency support need to be handled at application level only.

3.1.2 Programming for Persistent Memory

Persistent Memory (PM) unifies the memory and storage functionality by levering fast, byte-addressable, non-volatile memory technologies. PM technologies, such as Intel's Optane DC Persistent Memory [31], allows programs to manipulate the persistent data in memory via

memory instructions directly. Thus, without OS interventions, applications can efficiently leverage the PM system's high performance.

However, it brings the challenge of efficient system support, that was generally the file system's responsibility in conventional systems. As persistent data is expected to be recoverable in a crash, so PM systems have to provide support for crash-consistency at the application level [31]. This required lot of effort for the developers to create efficient data structures at application level. Therefore, programming the persistent memory considered as challenging and prone to errors.

3.1.3 Testing crash-consistency in PM Applications

For the crash-consistency during execution, applications need to ensure durability and ordering. In durability guarantee, it is enforced that data should reach the persistent medium reliably. Ordering guarantees ensure to order persistent operations explicitly because hardware can reorder instructions, so one writes must become persistent before another.

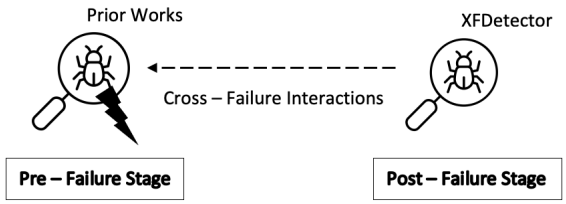

Figure 3.1 Prior works vs XFDetector [30]

3.1.4 Key Design Ideas for Cross-failure detection using XFDetector

The tool XFDetector, Xross-Failure Detector, traces PM operations in both pre- and post-failure stages. Here, we explained the approach to determine data consistency to detect cross-failure races and inject failures into the PM application to cover all cross failure interactions.

1. **Data consistency** To guarantee the data consistency across the failure, we need to verify if the data read by the post-failure stage is consistent. Data consistency depends on the program's manipulation of persistent data, so one needs to understand the data consistency based on its execution.

2. **Failure Injection Mechanism** To detect the cross failure bugs, XFDetector injects failures during the program execution to trigger the pre- and post-failure stages. One observation was that updates to PM are not guaranteed to be persisted until explicitly written back(e.g., using a persist_barrier). The Program that writes back to the PM before any future operations are known as the ordering point. As the updates before the persist_barrier can potentially be inconsistent, XFDetector injects a failure point to each of the persist_barriers and then test if the interactions across the failure are correct or not.

3.1.5 Modelling crash-consistency bugs as a Cross-failure Race using Intel Pin

As we mentioned earlier, existing works have focused on the correctness of the stage before failure by detecting bugs. However, crash consistency relies on stages both before and after the failure. This method holistically considers both pre- and post-failure execution stages. The following are the three stages of this method, which will demonstrate the mechanism of failure injection and bug detection together.

- **Purpose**: We surveyed earlier works to analyze the frameworks developed for evaluating the crash-consistency guarantees in software for PM applications [31][29]. We observed that these works focus only on the correctness of the stages before the failure. However, crash-consistency relies on stages before and after the failure occurs. Hence, we studied the work that defines cross failures interactions between the stages.

- **Detection Procedure using Intel Pin** The detection procedure consists of frontend that injects failures and traces PM operations using the Intel Pin tool and a back-end that detects bugs based on the traces. we will explain both parts below:

1. **Frontend**: We utilize the Intel Pin tool to perform tracing and failure injection. In this process, first, we locate all ordering points in the binary and then instruments the binary with failure handlers before each ordering point. After instrumentation, the frontend performs tracing and also failure injection during execution. On encountering a failure point within the Region-of-Interest(RoI), XFDetector suspends the program, creates a copy of the PM image(pool file on PM), and spawns its post-failure execution. Then XFDetector generates another trace until it reaches its termination point. This way, traces of both pre-failure and post-failure stages are collected.

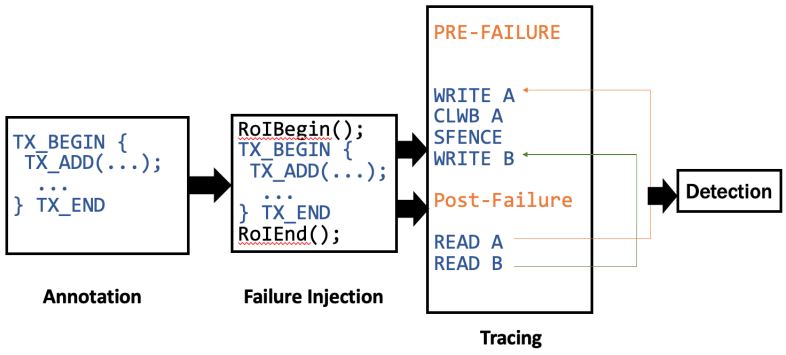

Figure 3.2 Detection mechanism in XFDetector

2. **Backend**: In this stage, XFDetector uses the PM state to detect the cross-failure race and consistency state to detect the cross-failure semantic bug. During the detection, XFDetector replays the traces in order of pre-failure and post-failure. On detection of cross-failure bug, XFDetector displays the program's file name and line number of reader(post-failure stage) and last writer(pre-failure stage) that causes the bug.

3.2 Experiment Methodology

This section describes our software and hardware platform to evaluate the bugs reproduced by XFDetector.

3.2.1 Evaluated System

We evaluated the XFDetector inside a virtual machine with emulated PM, where PM is mounted with the DAX file system to bypass OS indirections. Installation steps for XFDetector are available in B.2.

DRAM for VM	8GB
Guest OS	Ubuntu 18.04, linux kernel 4.15
PM	4GB(Emulated), ext4-DAX
Tool	Intel Pin 3.10(Tracing)
Libraries	gcc-7.4, PMDK-1.6, ndctl-61.2

Table 3.1 Evaluated platform

3.2.2 Persistent Memory Emulation

For our experiments, we emulated the persistent memory using DRAM allocated for the Virtual Machine(VM). Here, we introduce how we emulated the Persistent Memory block device (PMEM) support on the Linux VM.

We followed the below steps to allocate the space for persistent memory in our virtual machine:

- **Create PMEM device for persistent memory**

  First we identified the available physical addresses using

  dmesg — grep BIOS-e820

  From the output of above command, we decided to allocate the space of 4GB for persistent memory from 4G to 8G address range in the following 2 steps.

sudo vim /etc/default/grub

GRUB_CMDLINE_LINUX="memmap=4G!4G"

Save changes and execute

sudo update-grub

Then reboot the VM to make the changes take effect.After the restart, we see a new PMEM device with directory /dev/pmem0 on using command

df -Th

- **Create and Build a DAX-enabled file system**

  We created a mounting point with the name pmem

  mkdir /mnt/pmem

  We created an ext4 filesystem on /dev/pmem

  mkfs.ext4 /dev/pmem0

  mount -o dax /dev/pmem0 /mnt/pmem

  Now, we were able to use persistent memory as a regular file directory.

3.3 Results

This section will demonstrate the results of Cross-failure bug detection in persistent memory generated using the XFDetector tool. Later, we will also discuss some of the opportunities in optimizing the reliability of persistent memory programs.

3.3.1 Synthetic Bugs reported by XFDetector

Synthetic bugs are the bugs introduced intentionally to show the proof-of concept for the bug detection tool. This section will describe the crash-consistency bugs reported by the XFDetector tool in various PM applications.

- **Evaluation using PMDK Microbenchmarks**: Following bugs were reproduced by XFDetector in PMDK examples that were mainly missing protection by transaction while updating data structures.

1. **Cross-failure Race in Btree**:

   (a) Data Structure not protected by transaction: This buggy patch removes the transaction that back up the root field in the undo log.

```
diff --git a/src/examples/libpmemobj/tree_map/btree_map.c b/src/examples/libpmemobj/tree_map/btree_map.c
index f02a74a96..13a81b8e8 100644
--- a/src/examples/libpmemobj/tree_map/btree_map.c
+++ b/src/examples/libpmemobj/tree_map/btree_map.c
@@ -241,7 +241,8 @@ btree_map_find_dest_node(TOID(struct btree_map) map,
                         D_RW(up)->slots[0] = n;
                         D_RW(up)->slots[1] = right;
-                        TX_ADD_FIELD(map, root);
+                        // Remove TX_ADD_FIELD
+                        // TX_ADD_FIELD(map, root);
                         D_RW(map)->root = up;
                         n = up;
```

Figure 3.3 Buggy Patch to detect cross-failure race [16]

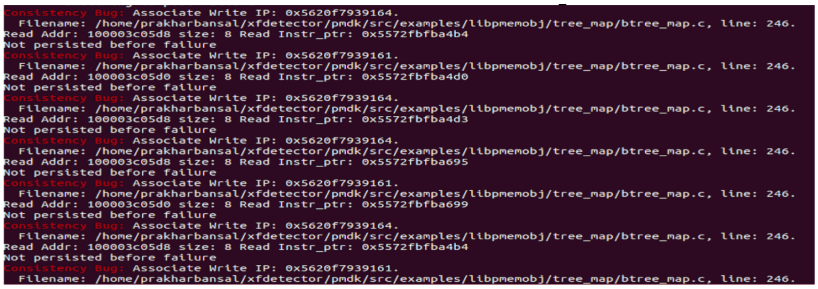

Figure 3.4 Cross-failure race detected in Btree

Fig 3.4 shows the Cross-failure race detected by XFDetector on line 246 due to missing TX_ADD function that adds PM object to undo log for recovery purposes during failure.

(b) Data Structure not protected by a transaction: In this patch, object node is not protected by transaction before it is updated.

```
diff --git a/src/examples/libpmemobj/tree_map/btree_map.c b/src/examples/libpmemobj/tree_map/btree_map.c
index f02a74a96..cafabe601 100644
--- a/src/examples/libpmemobj/tree_map/btree_map.c
+++ b/src/examples/libpmemobj/tree_map/btree_map.c
@@ -198,9 +198,12 @@ btree_map_create_split_node(TOID(struct tree_map_node) node,
        int c = (BTREE_ORDER / 2);
        *m = D_RO(node)->items[c - 1]; /* select median item */
-       TX_ADD(node);
+
        set_empty_item(&D_RW(node)->items[c - 1]);

+       // BUG: TX_ADD wrong place
+       TX_ADD(node);
+
        /* move everything right side of median to the new node */
        for (int i = c; i < BTREE_ORDER; ++i) {
                if (i != BTREE_ORDER - 1) {
```

Figure 3.5 Patch for detecting cross failure race [16]

Fig 3.5 shows the patch to detect another Cross-failure race due to missing instruction to persist the data before it is updated.

```
^[[0;31mConsistency Bug:^[[0m Associate Write IP: 0x556f829d133e.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map
/btree_map.c, line: 69.
Read Addr: 100003fe4e0 size: 8 Read Instr_ptr: 0x55cf955e5cdc
Not persisted before failure
^[[0;31mConsistency Bug:^[[0m Associate Write IP: 0x556f829d1349.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map
/btree_map.c, line: 70.
Read Addr: 100003fe4e8 size: 8 Read Instr_ptr: 0x55cf955e5ce0
Not persisted before failure
^[[0;31mConsistency Bug:^[[0m Associate Write IP: 0x556f829d1355.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map
/btree_map.c, line: 70.
Read Addr: 100003fe4f0 size: 8 Read Instr_ptr: 0x55cf955e5ceb
Not persisted before failure
-------Switching to Pre failure--------
```

Figure 3.6 Cross failure Bug detected by XFDetector

Above fig 3.6 shows the Cross-failure race detected at line 70 because of missing TX_ADD function to store the PM object in undo log.

(c) In the following patch, TX_ADD that backs up PM object node in undo log is removed.

```
diff --git a/src/examples/libpmemobj/tree_map/btree_map.c b/src/examples/libpmemobj/tree_map/btree_map.c
index f02a74a96..b3bcf67e5 100644
--- a/src/examples/libpmemobj/tree_map/btree_map.c
+++ b/src/examples/libpmemobj/tree_map/btree_map.c
@@ -198,7 +198,8 @@ btree_map_create_split_node(TOID(struct tree_map_node) node,
        int c = (BTREE_ORDER / 2);
        *m = D_RO(node)->items[c - 1]; /* select median item */
-       TX_ADD(node);
+       // BUG: Remove TX_ADD
+       //TX_ADD(node);
        set_empty_item(&D_RW(node)->items[c - 1]);
        /* move everything right side of median to the new node */
```

Figure 3.7 Btree cross failure race [16]

```
-------Switching to post failure--------
Consistency Bug: Associate Write IP: 0x5626b32edf0e.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/btree_map.c, line: 214.
Read Addr: 100003fe490 size: 4 Read Instr_ptr: 0x561cacc1b4e5
Not persisted before failure
Consistency Bug: Associate Write IP: 0x5626b32edf0e.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/btree_map.c, line: 214.
Read Addr: 100003fe490 size: 4 Read Instr_ptr: 0x561cacc1af93
Not persisted before failure
Consistency Bug: Associate Write IP: 0x5626b32edf0e.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/btree_map.c, line: 214.
Read Addr: 100003fe490 size: 4 Read Instr_ptr: 0x561cacc1b1b2
Not persisted before failure
Consistency Bug: Associate Write IP: 0x5626b32edf0e.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/btree_map.c, line: 214.
Read Addr: 100003fe490 size: 4 Read Instr_ptr: 0x561cacc1b1b2
Not persisted before failure
Consistency Bug: Associate Write IP: 0x5626b32edf0e.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/btree_map.c, line: 214.
Read Addr: 100003fe490 size: 4 Read Instr_ptr: 0x561cacc1b1b2
Not persisted before failure
```

Figure 3.8 Cross failure Bug detected in Btree

Fig 3.8 shows the cross failure race detected in Btree at line 214(after the Patch is applied) where post failure stage reads from non-persistent PM location written during pre-failure stage.

(d) The Following Patch in figure 3.9 removes the TX_ADD_FIELD that enforces write-back on the PM object's field.

```
diff --git a/src/examples/libpmemobj/tree_map/btree_map.c b/src/examples/libpmemobj/tree_map/btree_map.c
index f02a74a96..8de7bbf92 100644
--- a/src/examples/libpmemobj/tree_map/btree_map.c
+++ b/src/examples/libpmemobj/tree_map/btree_map.c
@@ -160,7 +160,8 @@ static void
 btree_map_insert_empty(TOID(struct btree_map) map,
        struct tree_map_node_item item)
 {
-       TX_ADD_FIELD(map, root);
+       // BUG: Remove TX_ADD_FIELD
+       //TX_ADD_FIELD(map, root);
        D_RW(map)->root = TX_ZNEW(struct tree_map_node);
        btree_map_insert_item_at(D_RO(map)->root, 0, item);
```

Figure 3.9 Patch for detecting cross failure race in Btree

```
Consistency Bug: Associate Write IP: 0x5571bd4189a8.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/btree_map.c, line: 164.
Read Addr: 100003c05d8 size: 8 Read Instr_ptr: 0x55f8c67504b4
Not persisted before failure
Consistency Bug: Associate Write IP: 0x5571bd4189a5.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/btree_map.c, line: 164.
Read Addr: 100003c05d0 size: 8 Read Instr_ptr: 0x55f8c67504d0
Not persisted before failure
Consistency Bug: Associate Write IP: 0x5571bd4189a8.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/btree_map.c, line: 164.
Read Addr: 100003c05d8 size: 8 Read Instr_ptr: 0x55f8c67504d3
Not persisted before failure
Consistency Bug: Associate Write IP: 0x5571bd4189a8.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/btree_map.c, line: 164.
Read Addr: 100003c05d8 size: 8 Read Instr_ptr: 0x55f8c6750695
Not persisted before failure
Consistency Bug: Associate Write IP: 0x5571bd4189a5.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/btree_map.c, line: 164.
Read Addr: 100003c05d0 size: 8 Read Instr_ptr: 0x55f8c6750699
Not persisted before failure
```

Figure 3.10 Cross failure bug detected in Btree

Fig 3.10 shows the cross failure race detected at line 164 in Btree.

2. **Cross-failure race in Ctree**: Figure 3.11 shows the buggy patch in Ctree where pmem_obj_tx_add_range function was removed which takes the snapshot of memory block for object p.

```
diff --git a/src/examples/libpmemobj/tree_map/ctree_map.c b/src/examples/libpmemobj/tree_map/ctree_map.c
index 639da18d0..eb772c1f2 100644
--- a/src/examples/libpmemobj/tree_map/ctree_map.c
+++ b/src/examples/libpmemobj/tree_map/ctree_map.c
@@ -220,8 +220,10 @@ ctree_map_insert(PMEMobjpool *pop, TOID(struct ctree_map) map,
        struct tree_map_entry e = {key, value};
        TX_BEGIN(pop) {
                if (p->key == 0 || p->key == key) {
-                       pmemobj_tx_add_range_direct(p, sizeof(*p));
+                       // Misplace TX_ADD
+                       //pmemobj_tx_add_range_direct(p, sizeof(*p));
                        *p = e;
+                       pmemobj_tx_add_range_direct(p, sizeof(*p));
                } else {
                        ctree_map_insert_leaf(&D_RW(map)->root, e,
                                              find_crit_bit(p->key, key));
-
```

Figure 3.11 Buggy patch in Ctree [16]

```
Consistency Bug: Associate Write IP: 0x55568ff09833.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/ctree_map.c, line: 225.
Read Addr: 100003c05e0 size: 8 Read Instr_ptr: 0x564e6abf7708
Not persisted before failure
Consistency Bug: Associate Write IP: 0x55568ff09828.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/ctree_map.c, line: 225.
Read Addr: 100003c05d8 size: 8 Read Instr_ptr: 0x564e6abf7718
Not persisted before failure
Consistency Bug: Associate Write IP: 0x55568ff09833.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/ctree_map.c, line: 225.
Read Addr: 100003c05e0 size: 8 Read Instr_ptr: 0x564e6abf771c
Not persisted before failure
Consistency Bug: Associate Write IP: 0x55568ff09825.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/ctree_map.c, line: 225.
Read Addr: 100003c05d0 size: 8 Read Instr_ptr: 0x564e6abf77f5
Not persisted before failure
Consistency Bug: Associate Write IP: 0x55568ff09825.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/ctree_map.c, line: 225.
Read Addr: 100003c05d0 size: 8 Read Instr_ptr: 0x564e6abf7804
Not persisted before failure
Consistency Bug: Associate Write IP: 0x55568ff09825.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/ctree_map.c, line: 225.
```

Figure 3.12 Cross-failure race detected in Ctree

Figure 3.12 shows the bug detected by XFDetector at line 225 after applying the buggy path of 3.11.

3. **Cross-failure race in Rbtree**:

```
diff --git a/src/examples/libpmemobj/tree_map/rbtree_map.c b/src/examples/libpmemobj/tree_map/rbtree_map.c
index 48a4fe671..26484381d 100644
--- a/src/examples/libpmemobj/tree_map/rbtree_map.c
+++ b/src/examples/libpmemobj/tree_map/rbtree_map.c
@@ -229,8 +229,8 @@ rbtree_map_insert_bst(TOID(struct rbtree_map) map, TOID(struct tree_map_node) n)
         }
         TX_SET(n, parent, parent);
-
-        pmemobj_tx_add_range_direct(dst, sizeof(*dst));
+        // Missing tx add
+        //pmemobj_tx_add_range_direct(dst, sizeof(*dst));
         *dst = n;
 }
```

Figure 3.13 Buggy Patch for detecting cross failure race [16]

Figure 3.13 shows the buggy patch in Rbtree where pmem_obj_tx_add_range function was removed which takes the snapshot of memory block for object dst.

```
Consistency Bug: Associate Write IP: 0x55879b5e5a41.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/rbtree_map.c, line: 234.
Read Addr: 100003c0808 size: 8 Read Instr_ptr: 0x55e39d7fc9bb
Not persisted before failure
Consistency Bug: Associate Write IP: 0x55879b5e5a41.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/rbtree_map.c, line: 234.
Read Addr: 100003c0808 size: 8 Read Instr_ptr: 0x55e39d7fc93e
Not persisted before failure
Consistency Bug: Associate Write IP: 0x55879b5e5a3e.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/rbtree_map.c, line: 234.
Read Addr: 100003c0800 size: 8 Read Instr_ptr: 0x55e39d7fc942
Not persisted before failure
Consistency Bug: Associate Write IP: 0x55879b5e5a3e.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/rbtree_map.c, line: 234.
Read Addr: 100003c0800 size: 8 Read Instr_ptr: 0x55e39d7fc959
Not persisted before failure
Consistency Bug: Associate Write IP: 0x55879b5e5a41.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/rbtree_map.c, line: 234.
Read Addr: 100003c0808 size: 8 Read Instr_ptr: 0x55e39d7fc95c
Not persisted before failure
Consistency Bug: Associate Write IP: 0x55879b5e5a3e.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/tree_map/rbtree_map.c, line: 234.
Read Addr: 100003c0800 size: 8 Read Instr_ptr: 0x55e39d7fc988
Not persisted before failure
```

Figure 3.14 Cross-failure race detected in RBtree

Figure 3.14 shows the bug detected by XFDetector at line 234 after applying the buggy path of 3.13.

4. **Cross-failure race in hashmap_tx**:

   (a) In the following buggy patch, TX_ADD_FIELD transaction that is responsible for protecting the value of count is removed. So, the pre-failure stage may not be having a persisted value of count.

```
diff --git a/src/examples/libpmemobj/hashmap/hashmap_tx.c b/src/examples/libpmemobj/hashmap/hashmap_tx.c
index 02349d7f0..e45f89fc7 100644
--- a/src/examples/libpmemobj/hashmap/hashmap_tx.c
+++ b/src/examples/libpmemobj/hashmap/hashmap_tx.c
@@ -202,7 +202,8 @@ hm_tx_insert(PMEMobjpool *pop, TOID(struct hashmap_tx) hashmap,
        int ret = 0;
        TX_BEGIN(pop) {
                TX_ADD_FIELD(D_RO(hashmap)->buckets, bucket[h]);
-               TX_ADD_FIELD(hashmap, count);
+               // BUG: Missing TX_ADD_FIELD
+               // TX_ADD_FIELD(hashmap, count);
                TOID(struct entry) e = TX_NEW(struct entry);
                D_RW(e)->key = key;
```

Figure 3.15 Buggy Patch for detecting cross-failure race in Hashmap [16]

```
Consistency Bug: Associate Write IP: 0x5563ca26b7c2.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_tx.c, line: 214.
Read Addr: 100003c05e8 size: 8 Read Instr_ptr: 0x55643d24b7ba
Not persisted before failure
Post-failure time: 8678ms
--------Switching to Pre failure--------
Failure point IP: 0x5563ca25a5a8
--------Switching to post failure--------
Consistency Bug: Associate Write IP: 0x5563ca26b7c2.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_tx.c, line: 214.
Read Addr: 100003c05e8 size: 8 Read Instr_ptr: 0x56324ac027ba
Not persisted before failure
Post-failure time: 8645ms
--------Switching to Pre failure--------
Failure point IP: 0x5563ca2b4fee
--------Switching to post failure--------
Consistency Bug: Associate Write IP: 0x5563ca26b7c2.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_tx.c, line: 214.
Read Addr: 100003c05e8 size: 8 Read Instr_ptr: 0x5626684297ba
Not persisted before failure
Post-failure time: 8876ms
--------Switching to Pre failure--------
Failure point IP: 0x5563ca2b5cc9
--------Switching to post failure--------
Consistency Bug: Associate Write IP: 0x5563ca26b7c2.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_tx.c, line: 214.
Read Addr: 100003c05e8 size: 8 Read Instr_ptr: 0x560f67bf77ba
Not persisted before failure
Post-failure time: 8553ms
```

Figure 3.16 Cross-failure race in Hashmap

(b) In this buggy patch too, TX_ADD_FIELD transaction that is responsible for protecting the value of next field is removed. So, the pre-failure stage may not be having a persisted value of count, and the post-failure stage would be reading the non-persisted value of field next.

```
diff --git a/src/examples/libpmemobj/hashmap/hashmap_tx.c b/src/examples/libpmemobj/hashmap/hashmap_tx.c
index 02349d7f0..c42745ead 100644
--- a/src/examples/libpmemobj/hashmap/hashmap_tx.c
+++ b/src/examples/libpmemobj/hashmap/hashmap_tx.c
@@ -155,9 +155,11 @@ hm_tx_rebuild(PMEMobjpool *pop, TOID(struct hashmap_tx) hashmap, size_t new_len)

                                D_RW(buckets_old)->bucket[i] = D_RO(en)->next;

-                               TX_ADD_FIELD(en, next);
                                D_RW(en)->next = D_RO(buckets_new)->bucket[h];
                                D_RW(buckets_new)->bucket[h] = en;
+
+                               // BUG: TX_ADD_FILED after update
+                               TX_ADD_FIELD(en, next);
                        }
```

Figure 3.17 Buggy Patch for detecting cross-failure race in Hashmap

```
Consistency Bug: Associate Write IP: 0x55c0ce41623e.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_tx.c, line: 158.
Read Addr: 10000382f68 size: 8 Read Instr_ptr: 0x55d80c41e1d5
Not persisted before failure
Consistency Bug: Associate Write IP: 0x55c0ce416242.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_tx.c, line: 158.
Read Addr: 10000382f70 size: 8 Read Instr_ptr: 0x55d80c41e1d9
Not persisted before failure
Post-failure time: 9115ms
--------Switching to Pre failure--------
Consistency Bug: TX_ADD after modification. Write IP: 55c0ce4605a5 Write Addr: 10000382f68
  Filename: /home/prakharbansal/xfdetector/pmdk/src/libpmemobj/tx.c, line: 1122.
Failure point IP: 0x55c0ce460cf3
--------Switching to post failure--------
Post-failure time: 8995ms
--------Switching to Pre failure--------
Consistency Bug: TX_ADD after modification. Write IP: 55c0ce4605a5 Write Addr: 100003c14e8
  Filename: /home/prakharbansal/xfdetector/pmdk/src/libpmemobj/tx.c, line: 1122.
Failure point IP: 0x55c0ce460cf3
```

Figure 3.18 Cross failure race in Hashmap

5. **Cross-failure semantic bugs in hashmap_atomic**:

   (a) Following figure 3.19 is the buggy patch to show that post-failure stage reads from an inconsistent version of the count_dirty variable.

```
diff --git a/src/examples/libpmemobj/hashmap/hashmap_atomic.c b/src/examples/libpmemobj/hashmap/hashmap_atomic.c
index 4e137d40a..b794aecdc 100644
--- a/src/examples/libpmemobj/hashmap/hashmap_atomic.c
+++ b/src/examples/libpmemobj/hashmap/hashmap_atomic.c
@@ -250,8 +250,6 @@ int
 hm_atomic_insert(PMEMobjpool *pop, TOID(struct hashmap_atomic) hashmap,
                 uint64_t key, PMEMoid value)
 {
-       XFDetector_addCommitVar(&D_RW(hashmap)->count_dirty, sizeof(D_RW(hashmap)->count_dirty));
-       //fprintf(stderr ,"adding commit var\n");
        TOID(struct buckets) buckets = D_RO(hashmap)->buckets;
        TOID(struct entry) var;

@@ -457,7 +455,7 @@ hm_atomic_init(PMEMobjpool *pop, TOID(struct hashmap_atomic) hashmap)
 {
        srand(D_RO(hashmap)->seed);

-       //fprintf(stderr ,"Initing\n");
+       XFDetector_addCommitVar(&D_RO(hashmap)->count_dirty, sizeof(D_RO(hashmap)->count_dirty));
        //XFDetector_addCommitVar(&D_RO(hashmap)->count, sizeof(D_RO(hashmap)->count));

        /* handle rebuild interruption */
@@ -483,7 +481,9 @@ hm_atomic_init(PMEMobjpool *pop, TOID(struct hashmap_atomic) hashmap)
        }

        /* handle insert or remove interruption */
-       if (D_RO(hashmap)->count_dirty) {
+       // if (D_RO(hashmap)->count_dirty) {
+       // BUG: Wrong condition
+       if (!D_RO(hashmap)->count_dirty) {
                printf("count dirty, recalculating\n");
                TOID(struct entry) var;
                TOID(struct buckets) buckets = D_RO(hashmap)->buckets;
```

Figure 3.19 Buggy Patch for detecting bug in Hashmap_atomic

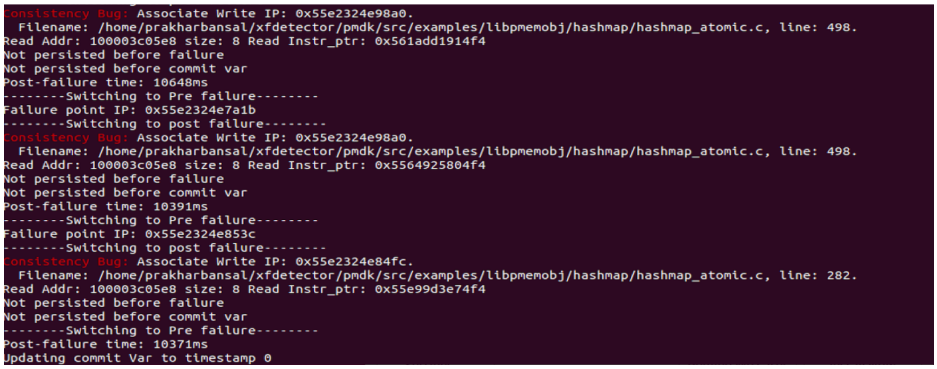

Figure 3.20 Cross failure semantic bug detected in Hashmap_atomic(line 498)

(b) The following buggy patch in figure 3.21 was used to show that post failure reads from inconsistent version of count.

```
diff --git a/src/examples/libpmemobj/hashmap/hashmap_atomic.c b/src/examples/libpmemobj/hashmap/hashmap_atomic.c
index 4e137d40a..b453f173a 100644
--- a/src/examples/libpmemobj/hashmap/hashmap_atomic.c
+++ b/src/examples/libpmemobj/hashmap/hashmap_atomic.c
@@ -264,7 +264,7 @@ hm_atomic_insert(PMEMobjpool *pop, TOID(struct hashmap_atomic) hashmap,
                num++;
        }

-       D_RW(hashmap)->count_dirty = 1;
+       D_RW(hashmap)->count_dirty = 0;
        pmemobj_persist(pop, &D_RW(hashmap)->count_dirty,
                        sizeof(D_RW(hashmap)->count_dirty));
```

Figure 3.21 Patch for detecting bug in Hashmap_atomic

```
Consistency Bug: Associate Write IP: 0x55ba8232652e.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_atomic.c, line: 284.
Read Addr: 100003c05e8 size: 8 Read Instr_ptr: 0x560021924526
Not persisted before failure
Not persisted before commit var
Post-failure time: 8742ms
--------Switching to Pre failure--------
Updating commit Var to timestamp 0
Failure point IP: 0x55ba823265d2
--------Switching to post failure--------
Consistency Bug: Associate Write IP: 0x55ba8232652e.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_atomic.c, line: 284.
Read Addr: 100003c05e8 size: 8 Read Instr_ptr: 0x55c79bf96526
Not persisted before failure
Not persisted before commit var
Post-failure time: 8795ms
--------Switching to Pre failure--------
Failure point IP: 0x55ba823175c6
--------Switching to post failure--------
Consistency Bug: Associate Write IP: 0x55ba8232652e.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_atomic.c, line: 284.
Read Addr: 100003c05e8 size: 8 Read Instr_ptr: 0x55df83d58526
Not persisted before failure
Not persisted before commit var
Post-failure time: 8721ms
--------Switching to Pre failure--------
Failure point IP: 0x55ba82372036
--------Switching to post failure--------
```

Figure 3.22 Cross failure semantic bug detected in Hashmap_atomic(line 284)

(c) The buggy patch in figure 3.23 was introduced to show that post-failure stage reads from inconsistent version of count.

```
diff --git a/src/examples/libpmemobj/hashmap/hashmap_atomic.c b/src/examples/libpmemobj/hashmap/hashmap_atomic.c
index 4e137d40a..07440f84a 100644
--- a/src/examples/libpmemobj/hashmap/hashmap_atomic.c
+++ b/src/examples/libpmemobj/hashmap/hashmap_atomic.c
@@ -285,7 +285,7 @@ hm_atomic_insert(PMEMobjpool *pop, TOID(struct hashmap_atomic) hashmap,
        pmemobj_persist(pop, &D_RW(hashmap)->count,
                        sizeof(D_RW(hashmap)->count));

-       D_RW(hashmap)->count_dirty = 0;
+       D_RW(hashmap)->count_dirty = 1;
        pmemobj_persist(pop, &D_RW(hashmap)->count_dirty,
                        sizeof(D_RW(hashmap)->count_dirty));
```

Figure 3.23 Buggy Patch in Hashmap_atomic [16]

```
Consistency Bug: Associate Write IP: 0x55639acfe900.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_atomic.c, line: 502.
Read Addr: 100003c05f0 size: 4 Read Instr_ptr: 0x5608866f16dd
Not persisted before failure
Consistency Bug: Associate Write IP: 0x55639acfe8ae.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_atomic.c, line: 498.
Read Addr: 100003c05e8 size: 8 Read Instr_ptr: 0x5608866f0526
Not persisted before failure
--------Switching to Pre failure--------
Post-failure time: 8992ms
Failure point IP: 0x55639ad49036
--------Switching to post failure--------
Consistency Bug: Associate Write IP: 0x55639acfe900.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_atomic.c, line: 502.
Read Addr: 100003c05f0 size: 4 Read Instr_ptr: 0x55ae460fd6dd
Not persisted before failure
Consistency Bug: Associate Write IP: 0x55639acfe8ae.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_atomic.c, line: 498.
Read Addr: 100003c05e8 size: 8 Read Instr_ptr: 0x55ae460fc526
Not persisted before failure
Post-failure time: 8688ms
```

Figure 3.24 Cross-failure semantic bug detected in Hashmap_atomic

3.3.2 Real Bugs reported by XFDetector

This section describes the bugs reproduced using the XFDetector in the real bugs detected in two applications; Redis and PMDK's Hashmap_atomic.

- **Cross failure race in Redis**: The following code snippet from Redis(server.c) [28] shows that the initialization procedure for num_dict_entries is not protected by a transaction. Thus, a failure in the middle of the initialization process can lead to a cross-failure race. To detect this issue, the source code of Redis was annotated(10 lines) using XFDetector's APIs.

```
void initPersistentMemory(void) {
    PMEMoid oid;
    struct redis_pmem_root *root;


    long long start = ustime();
    char pmfile_hmem[64];
    bytesToHuman(pmfile_hmem, server.pm_file_size);
    serverLog(LL_NOTICE,"Start init Persistent memory file %s size %s",
            server.pm_file_path, pmfile_hmem);

    /* Create new PMEM pool file. */
    server.pm_pool = pmemobj_create(server.pm_file_path, PM_LAYOUT_NAME, server.pm_file_size, 0666);

    if (server.pm_pool == NULL) {
        /* Open the existing PMEM pool file. */
        server.pm_pool = pmemobj_open(server.pm_file_path, PM_LAYOUT_NAME);
        server.pm_rootoid = POBJ_ROOT(server.pm_pool, struct redis_pmem_root);
        server.pm_reconstruct_required = true;

        if (server.pm_pool == NULL) {
            serverLog(LL_WARNING,"Cannot init persistent memory poolset file "
                "%s size %s", server.pm_file_path, pmfile_hmem);
            exit(1);
        }
    } else {
        server.pm_rootoid = POBJ_ROOT(server.pm_pool, struct redis_pmem_root);
        root = pmemobj_direct(server.pm_rootoid.oid);
        root->num_dict_entries = 0;
    }

    /* Get pool UUID from root object's OID. */
    oid = pmemobj_root(server.pm_pool, 1);
    server.pool_uuid_lo = oid.pool_uuid_lo;
```

post-failure can read inconsistent num_dict_entries without protection by transaction

Figure 3.25 Inconsistent data in Redis-nvml

```
^[[0;31mConsistency Bug:^[[0m Associate Write IP: 0x55b933f3273e.
  Filename: /home/prakharbansal/xfdetector/redis-nvml/src/server.c, line: 4039.
Read Addr: 100003c0550 size: 8 Read Instr_ptr: 0x55d5517c3a9c
Not persisted before failure
~
```

Figure 3.26 Cross-failure race in Redis-nvml

Fig 3.26 shows the cross-failure race detected in Redis at line 4039 after applying the XFDetector's patch.

- **Cross-failure race in PMDKs' Hashmap_atomic**: These are the real bugs found by XFDetector in Hashmap_atomic.c:

  1. Following code snippet that the seed value and hash_fun_a were not protected by crash-consistency mechanism[23].

```
*/
static void
create_hashmap(PMEMobjpool *pop, TOID(struct hashmap_atomic) hashmap,
               uint32_t seed)
{
        D_RW(hashmap)->seed = seed;

        do {
                D_RW(hashmap)->hash_fun_a = (uint32_t)rand();
        } while (D_RW(hashmap)->hash_fun_a == 0);
        D_RW(hashmap)->hash_fun_b = (uint32_t)rand();
        D_RW(hashmap)->hash_fun_p = HASH_FUNC_COEFF_P;

        size_t len = INIT_BUCKETS_NUM;
        size_t sz = sizeof(struct buckets) +
                        len * sizeof(struct entries_head);

        if (POBJ_ALLOC(pop, &D_RW(hashmap)->buckets, struct buckets, sz,
                        create_buckets, &len)) {
                fprintf(stderr, "root alloc failed: %s\n", pmemobj_errormsg());
                abort();
        }

        pmemobj_persist(pop, D_RW(hashmap), sizeof(*D_RW(hashmap)));
}
```

Figure 3.27 Non-persisted Data in Hashmap_atomic.c:(132-138)

Following figure 3.28 shows the cross-failure race detected by XFDetector as post-failure reads from invalid seed value and function pointers that were not completely persisted to PM in pre-failure stage.

```
--------Switching to post failure--------
^[[0;31mConsistency Bug:^[[0m Associate Write IP: 0x55eb622a8af0.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_atomic.c, line: 134.
Read Addr: 100003c05d0 size: 4 Read Instr_ptr: 0x55590b5374ba
Not persisted before failure
^[[0;31mConsistency Bug:^[[0m Associate Write IP: 0x55eb622a8b16.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_atomic.c, line: 137.
Read Addr: 100003c05d4 size: 4 Read Instr_ptr: 0x55590b535c96
Not persisted before failure
^[[0;31mConsistency Bug:^[[0m Associate Write IP: 0x55eb622a8b5f.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_atomic.c, line: 139.
Read Addr: 100003c05d8 size: 4 Read Instr_ptr: 0x55590b535cb2
Not persisted before failure
^[[0;31mConsistency Bug:^[[0m Associate Write IP: 0x55eb622a8b87.
  Filename: /home/prakharbansal/xfdetector/pmdk/src/examples/libpmemobj/hashmap/hashmap_atomic.c, line: 140.
Read Addr: 100003c05e0 size: 8 Read Instr_ptr: 0x55590b535cce
Not persisted before failure
```

Figure 3.28 Cross-failure race in Hashmap_atomic

2. Following code snippet from hasmhmap_atomic.c show that cross-failure race as post_failure stage can read from unmodified PM location(count) in the pre-failure stage.

```
int
hm_atomic_insert(PMEMobjpool *pop, TOID(struct hashmap_atomic) hashmap,
                uint64_t key, PMEMoid value)
{
        XFDetector_addCommitVar(&D_RW(hashmap)->count_dirty, sizeof(D_RW(hashmap)->count_dirty));
        //fprintf(stderr ,"adding commit var\n");
        TOID(struct buckets) buckets = D_RO(hashmap)->buckets;
        TOID(struct entry) var;

        uint64_t h = hash(&hashmap, &buckets, key);
        int num = 0;

        POBJ_LIST_FOREACH(var, &D_RO(buckets)->bucket[h], list) {
                if (D_RO(var)->key == key)
                        return 1;
                num++;
        }

        D_RW(hashmap)->count_dirty = 1;
        pmemobj_persist(pop, &D_RW(hashmap)->count_dirty,
                        sizeof(D_RW(hashmap)->count_dirty));

        struct entry_args args;
        args.key = key;
        args.value = value;

        PMEMoid oid = POBJ_LIST_INSERT_NEW_HEAD(pop,
                        &D_RW(buckets)->bucket[h],
                        list, sizeof(struct entry), create_entry, &args);
        if (OID_IS_NULL(oid)) {
                fprintf(stderr, "failed to allocate entry: %s\n",
                        pmemobj_errormsg());
                return -1;
        }

        D_RW(hashmap)->count++;
        pmemobj_persist(pop, &D_RW(hashmap)->count,
                        sizeof(D_RW(hashmap)->count));

        D_RW(hashmap)->count_dirty = 0;
        pmemobj_persist(pop, &D_RW(hashmap)->count_dirty,
```

Figure 3.29 Uninitialized PM location(count)

The Following figure 3.30 shows the bug detected by XFDetector where data was not persisted after pre-failure stage.

Figure 3.30 Cross-failure race in Hashmap_atomic.c

3.4 Opportunities to Optimize Reliability in PM systems

- **Automating the Process of Identifying Ordering Issues**: Earlier works described above require developers to annotate the source code and generate extensive test suites to test PM applications thoroughly. Finding persistency bugs can be automated by identifying application-independent patterns with missing or extra flush/fence instructions. Only application-specific bugs such as transaction misuse while updating data structures cannot be identified using this method.

- **Symbolic execution to expand Path coverage**: To cover all the possible executable paths of the PM application, symbolic execution tools such as KLEE [21] can be useful. A new tool can be built based on a symbolic model that can detect the persistency in the PM application without accessing the underlying PM resources.

- **Reducing run time overhead with Static Analysis**: Dynamic Analysis tool such as Intel Pin that is used for tracing in XFDetector causes high run time overhead, which leads to high execution time for the detection tool. To reduce such overhead, static analysis tools such as LLVM [8] are useful, where using LLVM's module pass, we can iterate over all the program's functions and then iterate over store instructions to identify ordering issues in PM application.

CHAPTER 4. Conclusion and Future work

Key-value stores such as RocksDB [24], Memcached [34], Tao [20] leverages large DRAM to achieve high throughput and low latency. Increasing DRAM may not improve performance because of cell sizes, cost, DIMM slot availability, etc. Therefore, Researchers are working to build systems that exploit emerging NVM technologies to minimize the performace and density gap between memory and storage. However, the performance delivers by these systems to applications still not near to the performance in DRAM systems. The performance analysis in 2 demonstrates the need for large DRAM for the key-value store RocksDB [24] to deliver high performance to applications. The proposed opportunities based on emerging NVM technologies can also bridge the performance gap between DRAM and persistent storage(SSDs/HDDs).

In the future, we would like to work on proposed opportunities in performance optimization in RocksDB as described in chapter 2. Also, we intend to use the NVM as a caching layer for flash SSDs, where recently accessed data would be on NVM, and old data can be flushed to flash storage when the NVM is full in capacity. This can improve the performance of RocksDB and reduce write amplification and write stalls.

Since programming for PM requires developers to know low-level primitives and PM libraries, it is hard and prone to errors. Hence, efficient testing frameworks are necessary to evaluate crash-consistency guarantees in software developed to leverage PM, such as optimized file system(PMFS), Intel PMDK [11]. Reliability Analysis using cross-failure bug detection tool XFDetector [30] in chapter 3 shows the bugs detected in PMDK's microbenchmarks and real-world applications such as Redis [28]. Since XFDetector used Intel Pin to trace the pre- and post-failure stages, the tool incurs high overhead and also requires human effort to generate test cases to find persistency bugs. Therefore, in the future, we would like to work on proposed opportunities in 3 to improve the PM system's reliability support.

BIBLIOGRAPHY

[1] Cassandra. Available:. http://cassandra.apache.org/.

[2] Dragon, A distributed graph query. Available:. https://code.fb.com/data-infrastructure/dragon-a-distributed-graph-query-engine/.

[3] Facebook.RocksDB: Block Cache.URL. Available:. https://github.com/facebook/rocksdb/wiki/Block-Cache.

[4] Facebook.RocksDB:Leveled Compaction Strategy. Available:. https://github.com/facebook/rocksdb/wiki/Leveled-Compaction.

[5] Intel corporation 2015, Intel/Micron 3D-XPoint Non-voaltile Main Memory. Available:. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[6] Intel corporation 2019, Intel Optane DC persistent memory. Available:. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[7] LevelDB. Available:. http://leveldb.org/.

[8] LLVM Compiler Infrastructure. Available:. http://llvm.org.

[9] MongoRocks. Available:. https://github.com/mongodb-partners/mongo-rocks.

[10] MyRocks. Available:. https://http://myrocks.io/.

[11] PMDK. Available:. https://github.com/pmem/pmdk.

[12] pmemrocksDB. Available:. https://github.com/pmem/pmem-rocksdb.

[13] RocksDB. Available:. http://rocksdb.org/.

[14] RocksDB's techtalk. Available:. https://www.slideshare.net/HiveData/tech-talk-rocksdb-slides-by-dhruba-borthakur-haobo-xu-of-facebook.

[15] RocksDB's users and use cases. Available:. https://github.com/facebook/rocksdb/wiki/RocksDB-Users-and-Use-Cases.

[16] Xfdetector. Available:. https://github.com/sihangliu/xfdetector.

[17] YCSB on RocksDB. Available:. https://github.com/brianfrankcooper/YCSB/tree/master/rocksdb.

[18] Allen, B. S. An analysis of persistent memory use with whisper. *ACM SIGPLAN Notices 52*, 4 (2017), 135–148.

[19] Arulraj, J., and Pavlo, A. "How to build a non-volatile memory database management system". *In Proceedings of the ACM International Conference on Management of Data (SIGMOD)* (2017).

[20] Bronson, N., Z. Amsden, G. C., P. Chakka, P. D., H. Ding, J. F., A. Giardullo, S. K., H. Li, M. M., D. Petrov, L. P., and Y. J. Song, V. V. "TAO: Facebook's Distributed Data Store for the Social Graph". *In Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13), San Jose, CA* (2013), 49–60.

[21] Cadar, C., Dunbar, D., and Engler., D. R. "Klee: Unas- sisted and automatic generation of high-coverage tests for com- plex systems programs". *In Proceedings of the Usenix Symposium on Operating Systems design and Implementation* (2008).

[22] Chen, G.J., e. "Real-time data processing at Facebook". *In ACM SIGMOD* (2016), 1087–1098.

[23] Cooper, B., and et al. "Benchmarking Cloud Serving Systems with YCSB". *ACM Symposium on Cloud Computing (SoCC), Indianapolis, Indiana* (2010).

[24] Dong, S., Callaghan M., Galanis, L., Borthakur D., Savor, T., and M., S. "Optimizing space amplification in RocksDB". *8th Biennial Conference on Innovative Data Systems Research(CIDR* (2017).

[25] EISENMAN, A., GARDNER D., A. I., AXBOE J., D. S., HAZELWOOD K., P. C., and CIDON A., KATTI, S. "Reducing DRAM footprint with NVM in facebook". *In Proceedings of the Thirteenth EuroSys Conference. ACM*, 42 (2018).

[26] Kadekodi, R., SeKwonLee, S. K., and Taesoo Kim, V. C. "SplitFS: A File System that Minimizes Software Overhead in File Systems for Persistent Memory". *In Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP), Ontario, Canada* (2019).

[27] Kannan, S., Nitish Bhat, A. G., An-drea, A.-D., and Arpaci-Dusseau, R. "Redesigning lsms for nonvolatile memory with novelsm". *USENIX Annual Technical Conference (ATC 18)* (2018).

[28] Kaur, G., and Kaur, J. "in-memory data processing using redis database". *International Journal of Computer Applications 180* (03 2018), 26–31.

[29] Liu, S., A. Kolli, J. R., and Khan., S. "Crash consistency in encrypted non-volatile main memory systems". *HPCA 2018* (2018).

[30] Liu, S., Korakit Seemakhupt, Y. W., Thomas Wenisch, A. K., and Khan, S. "Cross- failure bug detection in persistent memory programs". *In Proceedings of the Twenty-Fifth International Con- ference on Architectural Support for Programming Languages and Operating Systems* (2020), 1187–1202.

[31] Liu, S., Yizhou Wei, J. Z., Kolli, A., and Khan, S. "PMTest: A fast and flexible testing framework for persistent memory programs". *ASPLOS* (2019).

[32] Lu, L., Thanumalayan Sankaranarayana Pillai, A. C. A.-D., and Arpaci-Dusseau., R. H. "WiscKey: Separating Keys from Values in SSD-conscious Storage.". *In 14th USENIX Conference on File and Storage Technologies (FAST 16), pages 133–148, Santa Clara, CA, February 2016. USENIX Association.* (2016).

[33] Matsunobu, Y., and Siying Dong, H. L. "MyRocks: LSM-tree database storage engine serving Facebook's social graph". *Proceedings of the VLDB Endowment 13*, 12 (2020), 3217–3230.

[34] Nishtala, R., H. Fugal, S. G., M. Kwiatkowski, H. L., H. C. Li, R. M., M. Paleczny, D. P., P. Saab, D. S., and T. Tung, V. V. "Scaling Memcache at Facebook". *In Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), Lombard,IL* (2013), 385–398.

[35] P., O., CHENG E., G. D., and E., O. "The log-structured merge-tree (LSM-tree)". *Acta Inf. 33*, 4 (1996), 351–385.

[36] YAO, T., ZHANG Y., W. J., CUI Q., T. L., JIANG H., XIE, C., and X., H. "Matrixkv: Reducing write stalls and write amplification in lsm-tree based KV stores with matrix container in NVM". *USENIX Annual Technical Conference (USENIX ATC 20)* (2020), 11–31.

[37] Z., C., DONG S., V. S., and H., D. D. "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook". *In USENIX FAST* (2020).

APPENDIX A. Source Codes

A.1 Enabling and configuring Block Cache in YCSB's source code for RocksDB

The following are code changes in RocksDB client for YCSB to configure the block cache:

```
--- a/rocksdb/src/main/java/site/ycsb/db/rocksdb/RocksDBClient.java
+++ b/rocksdb/src/main/java/site/ycsb/db/rocksdb/RocksDBClient.java
@@ -32,7 +32,7 @@ import java.util.concurrent.ConcurrentHashMap;
 import java.util.concurrent.ConcurrentMap;
 import java.util.concurrent.locks.Lock;
 import java.util.concurrent.locks.ReentrantLock;
-
+import org.rocksdb.util.SizeUnit;
 import static java.nio.charset.StandardCharsets.UTF_8;

 /**
@@ -97,21 +97,36 @@ public class RocksDBClient extends DB {
       Files.createDirectories(rocksDbDir);
     }

-    final DBOptions options = new DBOptions();
+    final Options opts = new Options();
+    final BlockBasedTableConfig tableConfig = new BlockBasedTableConfig();
+    final Cache cache = new LRUCache(8 * SizeUnit.MB);
+    tableConfig.setBlockCache(cache);
+    //opts.setTableFormatConfig(tableConfig);
+
+    final DBOptions options = new DBOptions(opts);
     final List<ColumnFamilyDescriptor> cfDescriptors = new ArrayList<>();
     final List<ColumnFamilyHandle> cfHandles = new ArrayList<>();

     RocksDB.loadLibrary();
     OptionsUtil.loadOptionsFromFile(optionsFile.toAbsolutePath().toString(), Env.getDefault(), options, cfDescriptors);
+
+    for(int i = 0; i < cfDescriptors.size(); i++) {
+      final ColumnFamilyOptions cfOptions = cfDescriptors.get(i).getOptions();
+      cfOptions.setTableFormatConfig(tableConfig);
+    }
+
     dbOptions = options;

     final RocksDB db = RocksDB.open(options, rocksDbDir.toAbsolutePath().toString(), cfDescriptors, cfHandles);

     for(int i = 0; i < cfDescriptors.size(); i++) {
       String cfName = new String(cfDescriptors.get(i).getName());
+      LOGGER.info("Column family: " + cfName);
       final ColumnFamilyHandle cfHandle = cfHandles.get(i);
       final ColumnFamilyOptions cfOptions = cfDescriptors.get(i).getOptions();
```

Figure A.1 Block cache configuration

A.2 Shell Scripts to track memory and swap space usage

Following scripts were used to track the memory usage for the YCSB process since RocksDB is an embedded database and we benchmark it with YCSB.

```
#!/bin/bash

TS=`date +%s`

OUT_OUT=rocksdb_$TS.txt

b=""
while [ -z $b ] || [ $b -ne 0 ]
do
  b=`pgrep -l "java" | awk '{print $1}'`
  if [ -z $b ]; then
    continue
  fi
  echo "PID: $b"
  ps -p $b -o pid,ppid,%cpu,%mem,rss,sz,vsz,time >> $OUT_OUT
  sleep 0.5
done
```

Figure A.2 Shell script for Memory usage tracking through "ps"

Following script was used to track the swap space usage along with the memory usage by the QEMU-KVM Virtual Machine processes.

```
#!/bin/bash

TS=`date +%s`

OUT_OUT=output_$TS.txt
PID=$1
b=0
while [ $b -eq 0 ]
do

  ps -p $PID  -o pid,ppid,%cpu,%mem,rss,sz,vsz,time >> $OUT_OUT
  b=`echo $?`
  echo "PID: $PID"
  grep --color VmSwap /proc/$PID/status  >> $OUT_OUT
  sleep 0.5
done
```

Figure A.3 Shell script for VMs swap space usage through "VmSwap"

A.3 RocksDB Options

```
[CFOptions "default"]
  sample_for_compression=0
  compaction_pri=kMinOverlappingRatio
  merge_operator=nullptr
  compaction_filter_factory=nullptr
  memtable_factory=SkipListFactory
  memtable_insert_with_hint_prefix_extractor=nullptr
  comparator=leveldb.BytewiseComparator
  target_file_size_base=67108864
  max_sequential_skip_in_iterations=8
  compaction_style=kCompactionStyleLevel
  max_bytes_for_level_base=536870912
  bloom_locality=0
  write_buffer_size=134217728
  compression_per_level=kNoCompression:kNoCompression:kSnappyCompression:kSnappyCompression:kSnappyCompression:kSnappyCompression:kSnappyCompression
  memtable_huge_page_size=0
  max_successive_merges=0
  arena_block_size=16777216
  memtable_whole_key_filtering=false
  target_file_size_multiplier=1
  max_bytes_for_level_multiplier_additional=1:1:1:1:1:1:1
  snap_refresh_nanos=500000000
  num_levels=7
  min_write_buffer_number_to_merge=4
  max_write_buffer_number_to_maintain=0
  max_write_buffer_number=6
  compression=kSnappyCompression
  level0_stop_writes_trigger=36
  level0_slowdown_writes_trigger=20
  compaction_filter=nullptr
  level0_file_num_compaction_trigger=2
  max_compaction_bytes=1677721600
  compaction_options_universal={stop_style=kCompactionStopStyleTotalSize;compression_size_percent=-1;allow_trivial_move=false;max_merge_width=4294967
295;max_size_amplification_percent=200;min_merge_width=2;size_ratio=1;}
```

Figure A.4 CF Options section for column family "default"

A.4 Configuration of YCSB Workload

```
# Yahoo! Cloud System Benchmark
# Workload A: Update heavy workload
#   Application example: Session store recording recent actions
#
#   Read/update ratio: 50/50
#   Default data size: 1 KB records (10 fields, 100 bytes each, plus key)
#   Request distribution: zipfian
fieldcount=1
fieldlength=256
recordcount=100000000
operationcount=100000000
workload=site.ycsb.workloads.CoreWorkload

readallfields=true

readproportion=0.5
updateproportion=0.5
scanproportion=0
insertproportion=0

requestdistribution=zipfian
```

Figure A.5 Workload A(update heavy) for record size of 256 bytes

APPENDIX B. Installation Procedures

This is now the same as any other chapter except that all sectioning levels below the chapter level must begin with the *-form of a sectioning command.

B.1 Steps for running YCSB on RocksDB

- **Set up YCSB**

  git clone https://github.com/brianfrankcooper/YCSB.git

  cd YCSB

  mvn clean package

- **Run YCSB**

  Workload Statistics in YCSB:

  Create the database with 100 million random inserted KV pairs of size 280 bytes each:

  Load the data

  ./bin/ycsb load rocksdb -s -P workloads/workloada -p rocksdb.dir=/tmp/ycsb-rocksdb-data -p rocksdb.optionsfile=ycsb-rocksdb-options.ini

  Run the workload

  ./bin/ycsb run rocksdb -s -P workloads/workloada -p rocksdb.dir=/tmp/ycsb-rocksdb-data -p rocksdb.optionsfile=ycsb-rocksdb-options.ini

- **RocksDB configuration parameters**

  rocksdb.dir -(required) A path to folder to hold the RocksDB data files.

  rocksdb.optionsfile- A path to RocksDB options file. Here the options file is present in YCSB's root directory where we running the workload.

B.2 Installation of XFDetector

- git clone https://github.com/sihangliu/xfdetector.git

  export PIN_ROOT= /home/prakhar/data/xfdetector/pin-3.10

  export PATH= $PATH:$PIN_ROOT

  export PMEM_MMAP_HINT=0x10000000000

  make

  PMEM_MMAP is a debugging functionality from PMDK that maps PM to a predefined virtual address. Proper execution of XFDetector requires that PMEM_MMAP_HINT, PIN_ROOT and PATH are set up.

B.3 Testing and Reproducing bugs using XFDetector

- **PMDK's Microbenchmarks**: Before running the program, run the following commands:

  export PIN_ROOT= /home/prakhar/data/xfdetector/pin-3.10

  export PATH= $PATH:$PIN_ROOT

  export PMEM_MMAP_HINT=0x10000000000

  Following are the detailed steps to reproduce the bugs available in [16]

  1. **Reproducing bugs in PMDK examples**: Follow the below steps to reproduced bug in PMDK's microbenchmarks:

     From root directory of xfdetector:

     cd xfdetector

     Then, user need to run script **run.sh** with the following command:

     ./run.sh WORKLOAD INITSIZE TESTSIZE [PATCH], where

     WORKLOAD is workload to test

     INITSIZE is number of data insertions

     TESTSIZE is number of additional data insertions when reproducing bugs with XFDetector

PATCH patch name that reproduces the bug

For example, Synthetic bug in btree with Patch btree_race1.sh can be reproduced by running:

./run.sh btree 5 5 race1

Similarly, other examples can be used to reproduce bugs by checking commands from /xfdetector/runallPMDK.sh from xfdetector's root.

2. **Real Bug in Redis**: Follow the below steps to reproduce the real bug in Redis:

   User need to run script **xfdetector/runRedis.sh**

   ./runRedis.sh TESTSIZE

   We use the TESTSIZE = 5 to reproduce the bug.